Server-Assisted Public-Key Cryptographic Method

FIELD OF THE INVENTION

The present invention relates to a server-assisted computational method for the RSA processing that is viable on the resource-constrained devices. The invention is relevant to the fields of client-server distributed computing and public-key cryptography.

BACKGROUND OF THE INVENTION

Public-key cryptography is proven effective as a mechanism for secure messaging in an open network where no intermediate routers are presumed trustworthy to the end-communicators. The RSA algorithm nowadays represents the most widely adopted public-key cryptographic algorithm.

The RSA core comprises of encoding and decoding modules that are primarily exponentiation engines. Suppose (e, n) constitutes the encoding key, the encryption process is an exponentiation of the message M being raised to the power e under the modulus n to give the cryptograph S. If (d, n) is the decoding key, the decryption is the process that raise S to the power d under the modulus n to recover the original message M.

The RSA technique exploits the un-surmountable complexity of discrete factorization to deter any attempts of cracking the key pair (e, d). The technique is thus safe for cryptographic purposes. Contemporarily, it forms the underpinning of many public-key infrastructure systems for e-business activities on the Internet.

As e-business is rapidly expanding to the users of wireless handhelds, such as mobile phones, a secure transaction protocol that is effective on the wireless domain is the most desired technology to the e-business practitioners in order for them to seamlessly extend the secure transaction activities from the wire-lined Internet to the wireless counterpart.

Nevertheless, the solution is not straightforward. Public-key cryptography is so much resource demanding that the technology has never been feasible on the resource-deprived computing devices, such as mobile handheld.

Interim solutions have been proposed which effect via reduction in security functionality or certificate fields in order to fit with the CPU limitation. The public-key infrastructure that prevails in the wire-lined world thus takes a reduced form, weaker functionality and security strength, when ported to the wireless domain.

WTLS has been proposed as such a streamlined form of the commonly employed SSL security protocol for the wireless world. A concern, however, is the incompatibility between the SSL and WTLS domains, resulting in a vulnerable gap at the wireless gateway and failing the most desired end-to-end secure message tunneling (Figure 1).

Prior art handles a similar problem of conducting the RSA crypto processing on an IC card with load sharing between the IC card and the host computer in a point-of-sales setup. In those methods, the encoding or decoding key that represents the secret parameter held inside the IC card is broken into bit blocks, e_0, e_1, e_2, ..., e_l.

$$e = e_0 + e_1 \cdot 2^k + e_2 \cdot 2^{2k} + \cdots + e_l \cdot 2^{lk}$$
$$M^e = (M)^{e_0} \cdot (M^{2^k})^{e_1} \cdot (M^{2^{2k}})^{e_2} \cdots (M^{2^{lk}})^{e_l} \mod n$$

The load sharing is done in the way that the host computer conducts the exponentiation for the base values of individual blocks (powers of 2^k, 2^{2k}, ..., 2^{lk} on M) whereas the IC card carries out the intra-block exponentiations (powers of e_0, e_1, e_2, ..., e_l) to obtain the final cryptograph M^e.

As the result, the secret key is well kept by withholding it in the IC card. The load sharing is effective. Nevertheless, the comment is that the computational requirement on the IC card is still significant.

The present invention employs a more powerful secrecy model and offloads more of the computational requirements to the server side. As a result, the processor-heavy RSA becomes practically possible on a resource-poor handheld device.

When the mobile handheld can act with the regular cryptographic capability, the need for a reduced security protocol, such as WTLS, is immaterial. Consequently, the mobile handheld can work in full compatibility with the existing Internet SSL protocol, and the end-to-end secure tunneling is possible (Figure 2).

SUMMARY OF THE INVENTION

The present invention is a client-server computing method to enable a resource-deprived device to accomplish the otherwise overwhelming public-key processing. It is made possible by shifting the load of computation to the powerful server computer on the Internet. The result is that the client device drives the resource-rich server computer to carry out the bulk of the computation for its sake. The merit is that the server during the process is totally blinded of the secret parameters (the message code and the crypto key) of the client.

The core of the RSA runtime is the exponentiation operation. During the encryption phase, a message code is numerically raised to the exponential power as specified by the encryption key. Upon decryption, the original message is recovered by another exponentiation using the decryption key on the cryptograph. The technique although computationally expensive, is mostly affordable to the Internet computers nowadays.

The present invention enables the handheld to leverage the computing power of the Internet server computer to bear the load of the exponentiation computation so that the public-key cryptography becomes possible on the handheld in a logical sense.

Our method employs a more powerful secrecy model in which the key is transformed and masked by a bunch of random numbers. Rather than withholding the long RSA key (1024 bits), the client can keep a portion of the data (128 bits) that correspond to the equivalent search space (2^{128}). With that, the load sharing can be attained much more effectively between the client and the server by offloading most of the exponentiation computation to the server side.

The present invention may be understood more fully by reference to the following detailed description and illustrative examples which are intended to exemplify non-limiting embodiments of the invention.

The first embodiment is a client-server scheme for the exponentiation operation.

The second embodiment extends on the robustness of the method. Intermediate results from the server side are cross-validated against one another to

discover and thus decline any sabotage attacks from the server side in the case that the server is compromised.

BRIEF DESCRIPTION OF DRAWINGS

Figure 1 illustrates the security weakspot at the wireless gateway.

Figure 2 shows the client-driven server-assisted strategy for the public-key cryptography.

Figure 3 is the flowchart showing the first embodiment of the present invention.

Figure 4 is the flowchart showing the second embodiment of the present invention.

DETAILED DESCRIPTION AND PREFERRED EMBODIMENT

The present invention will be more readily understood by referring to the following examples and preferred embodiments, which are given to illustrate the invention rather than limit its scope.

The present invention embodies two versions of design. The core of the RSA public-key cryptographic processing involves the computation of exponentiation operations. As the handheld device is incapable of carrying out the demanding processing, it ships the data and crypto parameters to the server computer and makes the server compute the exponentiations for it. The handheld, as the client in this relationship, ensures the privacy of his secret data and parameters by scrambling all the data he sends out to the server (Figure 2/01).

The server is totally blinded of the client's secrets. It takes the role of an exponentiation engine, producing the near-completion result for the cryptographic process (Figure 2/02). Upon returning of the exponentiation result, the handheld finishes off the entire computation with its unshared secrets to churn out the final cryptograph (Figure 2/03) for that cryptographic process. When communicating with the cryptograph, the handheld is guaranteed end-to-end security as no third party has the key to reveal the original message code.

In the similar process, the end-to-end security is achieved during the deciphering phase as well. A private message is sent to the handheld (Figure 2/04). The handheld as the client drives the server computer to carry out the exponentiation processing to arrive at a near-completed decryption result (Figure 2/05). Upon receiving the result, the handheld completes the decryption process with its unshared

secrets (Figure 2/06). Consequently, the most desired end-to-end communication model is secured.

In the following sections, the mathematical formulation and the communication protocols of the two embodiments are detailed.

EMBODIMENT 1

The first embodiment reformulates the RSA algorithm as a client-server computational scheme. In the scheme, the secret hiding for the message code and the client's crypto key is well considered.

As the formulation of the RSA algorithm is symmetric for both encryption and decryption, we simplify the discussion by posting the encryption case only. The resulting client-server scheme is also applicable for decryption case without modification.

A. Client-server model for Exponentiation

The goal is to shift to the server computer the load of calculating the cryptograph S from the message M and the crypto key e.

$$S = M^e \bmod n \qquad (1)$$

The exponent e is broken into components e_i, $i = 1,\ldots,k$.

$$\begin{aligned} e &= e_1(r_{12} - r_{11}) + \cdots + e_k(r_{k2} - r_{k1}) \\ S &= M^e = M^{e_1(r_{12}-r_{11})} \cdots\cdots M^{e_k(r_{k2}-r_{k1})} \end{aligned} \qquad (2)$$

The r_{ij} terms in (2) are integers of small-values. To hide M and e from the server, the client scrambles M and the e-components with random numbers. For $n=p\cdot q$, we have $\phi=(p-1)\cdot(q-1)$. Then

$$\tilde{M} = (a \cdot M) \bmod n \qquad (3)$$

$$\begin{aligned} e_{i1} &= (e_i + u_i \cdot r_{i1}) \bmod \phi \\ e_{i2} &= -(e_i + u_i \cdot r_{i2}) \bmod \phi \end{aligned} \quad ; i = 1,\cdots,k \qquad (4)$$

Define partial terms $z_{ij} = \tilde{M}^{e_{ij}} \bmod n$. Expand with (3) and (4),

$$z_{i1} = (a \cdot M)^{e_i + u_i \cdot r_{i1}} \bmod n$$
$$z_{i2} = (a \cdot M)^{-(e_i + u_i \cdot r_{i2})} \bmod n \qquad (5)$$

Solve (5) for $M^{e_i(r_{i2}-r_{i1})}$,

$$M^{e_i(r_{i2}-r_{i1})} = \left(a^{e_i(r_{i1}-r_{i2})} \cdot \left(z_{i1}{}^{r_{i2}} \cdot z_{i2}{}^{r_{i1}}\right)\right) \bmod n \quad (6)$$

Now, the last step follows the expression (2) and puts the k components as calculated in (6) together to derive the cryptograph S.

$$S = M^e = \left(A \cdot \prod_{i=1}^{k} \left(z_{i1}{}^{r_{i2}} \cdot z_{i2}{}^{r_{i1}}\right)\right) \bmod n \qquad (7)$$
$$\text{where} \quad a^e \cdot A \equiv 1 \pmod{n}$$

B. The client-server protocol

In a preprocessing phase, the client generates and stores in its memory the random numbers a, A. The job can be done by the client during its idle time or pre-computed by another computer and downloaded to the client in a secure channel. The actual implementation is flexible for this step.

During the runtime, the client generates the random decomposition of e as in (2,4), and scrambles the message M as in (3). The client then ships the data to the server where the partial terms z_{ij}'s are computed (as in (5)). Upon receiving the partial terms in return, the client computes (7) to obtain the cryptograph.

Referring to Figure 3, the client-server protocol is carried out in four steps:

1) Pre-processing (Figure 3/01)

The random number a and its reciprocal A are generated as the parameters for scrambling the message code (in (3)) before sending it to the server, and for de-scrambling for the final cryptograph after the partial terms have been returned from the server (in (7)).

2) Client generates random numbers (Figure 3/02)

The client generates a random decomposition of the crypto key e into a set of e_{ij} components. It is intended to ask the server to compute the partial terms of $M^{e_{ij}}$.

In order to hide the information from the server, the message code is scrambled with a to give $\widetilde{M}$, and the e_{ij} set is randomly re-ordered to give $\{\widetilde{e}_{ij}\}$.

With such scrambling and random-ordering, the server should have no easy way to guess out how the client derives the final cryptograph at the end.

The data $\widetilde{M}$, $\{\widetilde{e}_{ij}\}$ are then communicated to the server for the exponentiation computation.

3) Server computes exponentiations (Figure 3/03)

Upon receiving the scrambled data $\widetilde{M}$, $\{\widetilde{e}_{ij}\}$ from the client, the server calculates the exponentiation terms $\widetilde{z}_{ij}$ as in (5).

These $\widetilde{z}_{ij}$ partial terms are sent back to the client then.

4) Client derives cryptograph (Figure 3/04)

Having received the set of $\widetilde{z}_{ij}$ partial terms, the client reorders the set and extracts the relevant values for the z_{ij} terms. It then calculates the final cryptograph S as in (7).

C. Potential Attack is Minimum

Potential attack at this stage involves the guesswork for the r_{ij} values. Such attacks are extremely difficult to work out. If we choose $k = 11$, we have 22 r_{ij} terms. Even each term has a value no larger than 63, the search space for the guesswork is already as large as $63^{22} \cong 10^{39} \cong 2^{128}$, which would readily satisfy the security requirements of the nowadays Internet applications.

D. Efficiency Consideration

The computational burden for the client comes mostly from the calculation of (7). Eq. (7) requires modular exponentiations and multiplications. As commonly known, a batch of exponentiations can be carried out in a procedure of multiplications, and the number of multiplications is related to the bit length of the exponents and the number of exponentiations to be done in the batch.

By the above case of 22 exponentiations and each exponent is no larger than 63 (bit length is 6), the worst case would reckon roughly 132 modular multiplications and the average case is roughly 66.

In the comparison with the regular RSA, an exponentiation operation using a 1024-bit encoding key requires modular multiplications in the order of 2 times the encoding key length, i.e. 2048. Compared with that, the method by this embodiment presents a saving factor of 15 times or more to the client device on its CPU demand.

EMBODIMENT 2

This method extends the first embodiment on the robustness of the client-server model. The former method does not anticipate sabotage attacks from the server side. The client takes the server calculations to the final cryptograph result by Eq. (7) without hesitation.

However, in the case that the server were compromised, the client might subject to attacks of malicious data manipulation. Hacker on the server might forge the z_{ij} values either by manipulating the $\widetilde{M}$, $\{\widetilde{e}_{ij}\}$ data sent to the server, or might fake the z_{ij} values altogether.

This method curbs sabotage attacks by taking the server calculation through 2 iterations and cross-verifying the results to discover any happenings of server-side forgery.

A. 2-iteration model with cross-verification

Essentially, the method calculates M^e in 2 iterations of exponentiation. Forgery in any one of the iterations will get magnified in another. Without the knowledge of the client's secret parameters for those iterations, the attacker has no way to fake through the entire process.

The mathematical formulation is presented in the following. We decompose the exponent e (*ref.* (2)) with disparate parameters in 3 different formulations as follows.

$$
\begin{aligned}
e &= f_a \cdot g_a + h_a \\
&= (h_a + \varepsilon) \cdot g_b + h_b \\
&= f_a \cdot g_a + (h_a + \varepsilon) \cdot g_b + h_c
\end{aligned}
\qquad (2.1')
$$

$$
\begin{aligned}
M^e &= (M^{f_a})^{g_a} \cdot M^{h_a} \\
&= (M^{h_a} \cdot M^{\varepsilon})^{g_b} \cdot M^{h_b} \\
&= (M^{f_a})^{g_a} \cdot (M^{h_a} \cdot M^{\varepsilon})^{g_b} \cdot M^{h_c}
\end{aligned}
\qquad (2.2')
$$

And, the respective exponent terms, f_a, g_a, h_a, g_b, h_b, h_c, are decomposed like it was done in (2).

$$
\begin{aligned}
f_a &= f_{a1}(r_{a12} - r_{a11}) + \cdots + f_{ak}(r_{ak2} - r_{ak1}) \\
g_a &= g_{a1}(s_{a12} - s_{a11}) + \cdots + g_{ak}(s_{ak2} - s_{ak1}) \\
g_b &= g_{b1}(s_{b12} - s_{b11}) + \cdots + g_{bk}(s_{bk2} - s_{bk1}) \\
h_a &= h_{a1}(t_{a12} - t_{a11}) + \cdots + h_{ak}(t_{ak2} - t_{ak1}) \\
h_b &= h_{b1}(t_{b12} - t_{b11}) + \cdots + h_{bk}(t_{bk2} - t_{bk1}) \\
h_c &= h_{c1}(t_{c12} - t_{c11}) + \cdots + h_{ck}(t_{ck2} - t_{ck1})
\end{aligned}
\qquad (2.3')
$$

We scramble M in the same way as in (3) with the mask a.

$$
\tilde{M} = a \cdot M \bmod n \qquad (3)
$$

For the exponent terms, the random scrambling this time is done as follows. For $i = 1, \cdots, k$ and $j = 1, 2$:

$$
\begin{aligned}
f_{ai1} &= (f_{ai} + u_i \cdot r_{ai1}) & f_{ai2} &= -(f_{ai} + u_i \cdot r_{ai2}) \\
g_{ai1} &= (g_{ai} + v_{ai} \cdot s_{ai1}) & g_{ai2} &= -(g_{ai} + v_{ai} \cdot s_{ai2}) \\
g_{bi1} &= (g_{bi} + v_{bi} \cdot s_{bi1}) & g_{bi2} &= -(g_{bi} + v_{bi} \cdot s_{bi2}) \\
h_{ai1} &= (h_{ai} + w_{ai} \cdot t_{ai1}) & h_{ai2} &= -(h_{ai} + w_{ai} \cdot t_{ai2}) \\
h_{bi1} &= (h_{bi} + w_{bi} \cdot t_{bi1}) & h_{bi2} &= -(h_{bi} + w_{bi} \cdot t_{bi2}) \\
h_{ci1} &= (h_{ci} + w_{ci} \cdot t_{ci1}) & h_{ci2} &= -(h_{ci} + w_{ci} \cdot t_{ci2})
\end{aligned}
\qquad (4')
$$

In the 1st iteration, the z_{ij} terms are defined for the first-level exponentiation of (2.2') with respect to the exponent terms f_a, h_a, h_b and h_c.

$$
\begin{aligned}
z_{faij} &= \tilde{M}^{f_{ay}} \bmod n \\
z_{haij} &= \tilde{M}^{h_{ay}} \bmod n \\
z_{hbij} &= \tilde{M}^{h_{by}} \bmod n \\
z_{hcij} &= \tilde{M}^{h_{cy}} \bmod n
\end{aligned}
\qquad (5')
$$

These z_{ij} terms from (5') are combined to give the partial cryptographs, $\dot{S}_{fa}, \dot{S}_{ha}, \dot{S}_{hb}, \dot{S}_{hc}$, as defined in below.

$$\begin{aligned}
&\dot{S}_{fa} = b_f \cdot \tilde{M}^{f_a} \\
&\dot{S}_{ha} = b_h \cdot \tilde{M}^{h_a} \cdot \tilde{M}^{\varepsilon} \\
&\dot{S}_{hb} = \tilde{M}^{h_b} \\
&\dot{S}_{hc} = \tilde{M}^{h_c} \\
&\text{where} \quad \tilde{M}^{f_a} = \left(\prod_{i=1}^{k} \left(z_{fai1}{}^{r_{ai2}} \cdot z_{fai2}{}^{r_{ai1}} \right) \right) \bmod n \\
&\qquad\qquad \tilde{M}^{h_a} = \left(\prod_{i=1}^{k} \left(z_{hai1}{}^{t_{ai2}} \cdot z_{hai2}{}^{t_{ai1}} \right) \right) \bmod n \\
&\qquad\qquad \tilde{M}^{h_b} = \left(\prod_{i=1}^{k} \left(z_{hbi1}{}^{t_{bi2}} \cdot z_{hbi2}{}^{t_{bi1}} \right) \right) \bmod n \\
&\qquad\qquad \tilde{M}^{h_c} = \left(\prod_{i=1}^{k} \left(z_{hci1}{}^{t_{ci2}} \cdot z_{hci2}{}^{t_{ci1}} \right) \right) \bmod n
\end{aligned} \tag{7'}$$

Now in the 2nd iteration, the partial cryptographs are fed through the exponentiation process for one more time to complete (2.2') with the second-level exponentiation. We define another set of partial terms, $\dot{z}_{fij}, \dot{z}_{hij}$, for this iteration.

$$\begin{aligned}
&\dot{z}_{fij} = \dot{S}_{fa}{}^{g_{ay}} \bmod n \\
&\dot{z}_{hij} = \dot{S}_{ha}{}^{g_{by}} \bmod n
\end{aligned} \tag{8'}$$

Similar to (7'), the partial terms are combined to give the partial cryptographs.

$$\begin{aligned}
&\ddot{S}_1 = (\dot{S}_{fa})^{g_a} = \left(B_f \cdot \prod_{i=1}^{k} \left(\dot{z}_{fi1}{}^{s_{ai2}} \cdot \dot{z}_{fi2}{}^{s_{ai1}} \right) \right) \bmod n \\
&\ddot{S}_2 = (\dot{S}_{ha})^{g_b} = \left(B_h \cdot \prod_{i=1}^{k} \left(\dot{z}_{hi1}{}^{s_{bi2}} \cdot \dot{z}_{hi2}{}^{s_{bi1}} \right) \right) \bmod n \\
&\text{where} \quad b_f{}^{g_a} \cdot B_f \equiv 1 \,(\bmod n) \\
&\qquad\quad b_h{}^{g_b} \cdot B_h \equiv 1 \,(\bmod n)
\end{aligned} \tag{9'}$$

The final cryptograph S now can be derived with the partial cryptographs from (9'). From the formulation of (2.2'), three versions of S can be calculated.

$$\begin{aligned}
&S_1 = A \cdot \ddot{S}_1 \cdot \dot{S}_{ha} = (M^{f_a})^{g_a} \cdot M^{h_a} \bmod n \\
&S_2 = A \cdot \ddot{S}_2 \cdot \dot{S}_{hb} = (M^{h_a} \cdot M^{\varepsilon})^{g_b} \cdot M^{h_b} \bmod n \\
&S_3 = A \cdot \ddot{S}_1 \cdot \ddot{S}_2 \cdot \dot{S}_{hc} = (M^{f_a})^{g_a} \cdot (M^{h_a} \cdot M^{\varepsilon})^{g_b} \cdot M^{h_c} \bmod n
\end{aligned} \tag{10'}$$

The rationale for 3 different formulations for S is to build the mechanism in the process for cross-verification on the calculation of S. Agreement of the 3 versions indicates the validity of the server-side calculations. Hence, if $S_1 = S_2 = S_3$ in (10'), the calculations are considered to be correct, and any one of the three can be reported with confidence for the final cryptograph S.

B. The client-server protocol

1) Pre-processing (Figure 4/01)

Like it in the Embodiment 1, the random number a, and its reciprocal A, are generated as the parameters for scrambling the message code in (3), and for de-scrambling for the final cryptograph in (10').

In addition, two sets of random numbers, (g_a, b_f, B_f) and (g_b, b_h, B_h), are generated and stored in this pre-processing stage. The values g_a and g_b are to be used in (2.1') whereas (b_f, B_f) and (b_h, B_h) are the reciprocal pairs used in (7') and (9').

2) Client generates random numbers (Figure 4/02)

During runtime, the client generates the random decomposition of the crypto key e into the set of f_{aij} , h_{aij} , h_{bij} and h_{cij} terms (*ref.* (2') and (4')). Note that the ε in (2') as well as the r_{aij} , s_{aij} , s_{bij} , t_{aij} , t_{bij} and t_{cij} terms in (4') are all small integers such that the exponentiations with them by the client in the subsequent steps 4 and 6 are manageable.

The client scrambles M with a as in (3) to give $\tilde{M}$. The f_{aij} , h_{aij} , h_{cij} and h_{cij} terms are all mixed in one single pool and randomized in their ordering. Let the randomized sequence be referred as $\{\tilde{e}_{ij}\}$.

The scrambled $\tilde{M}$ and the randomized exponents $\{\tilde{e}_{ij}\}$ are sent to the server for computing the exponentiations.

3) Server computes exponentiations (Figure 4/03)

Upon receiving the scrambled data, $\tilde{M}$ and $\{\tilde{e}_{ij}\}$, from the client, the server calculates the exponentiation terms $\tilde{z}_{ij} = \tilde{M}^{\tilde{e}_{ij}}$.

4) Client calculates partial cryptographs
(Figure 4/04)

When the $\tilde{z}_{ij}$ partial terms are returned from the server side, the client undoes the random ordering of the set $\{\tilde{z}_{ij}\}$ to obtain the values for the respective terms of z_{faij}, z_{haij}, z_{hbij}, and z_{hcij}.

The client then calculates $\dot{S}_{fa}, \dot{S}_{ha}, \dot{S}_{hb}, \dot{S}_{hc}$ as in (7').

The client also calculates the decomposition of g_a and g_b for the sets of $\{g_{aij}\}$ and $\{g_{bij}\}$ (*ref.* (2') and (4')). Data of $(\dot{S}_{fa}, \{g_{aij}\})$ and $(\dot{S}_{ha}, \{g_{bij}\})$ are sent to the server for the 2nd iteration of exponentiation.

5) Server computes exponentiation of 2nd iteration
(Figure 4/05)

The server computes the $\dot{z}_{fij}$ values in (8') when $\dot{S}_{fa}, \{g_{aij}\}$ are received. By the same logic, it computes $\dot{z}_{hij}$ on the received data $\dot{S}_{ha}, \{g_{bij}\}$.

The results are then returned to the client side.

6) Client derives and verifies final cryptograph
(Figure 4/06)

The client derives the cryptograph in (9') and (10').

Three versions S_1, S_2 and S_3 are calculated. At this point, the client verifies the validity of these cryptographs against possible attacks from the server side by testing whether S_1, S_2 and S_3 all agree with each other. Testing positive, the client reports any one of the three as the final cryptograph $S = M^e$.

C. Verification Test is Effective

The verification test by the 2-iteration scheme is strong and tight in the sense that any malicious manipulation and forgery will be detected and prevented thereby.

Consider how the server-side attack could sabotage the overall calculation for $S = M^e$. Hacker breaking in the server could intercept the exponentiation processes as laid out in (5') and (8'). These calculations are in the form of $Z = X^Y$. Hence, the hacker could launch any of the following 3 attacks:

1. Manipulating X
2. Manipulating Y
3. Forging Z

1st form of Attack – Manipulating X.

The hacker could manipulate the $\widetilde{M}$ value in (5'), and thus faked the values for $\widetilde{M}^{f_a}, \widetilde{M}^{h_a}, \widetilde{M}^{h_b}, \widetilde{M}^{h_c}$ in (10'). Note that the calculation of $\widetilde{M}^{\varepsilon}$ is kept to the client side, and thus is safe from attacks. As the hacker has no way to estimate the impact of $\widetilde{M}^{\varepsilon}$ in the equation system (10'), he cannot manipulate $\widetilde{M}$ in such a way that the effect is coherent across S_1, S_2 and S_3. Hence, such attack is difficult.

2nd form of Attack – Manipulating Y.

The hacker could manipulate the exponents f_a, h_a, h_b, h_c and g_a, g_b by forging their values in the calculations of (5') and (8'). However, any manipulation on f_a and h_a will get magnified by the factors of g_a and g_b in the 2nd iteration, which are unknown to the hacker throughout the process. Therefore, the hacker indeed has no way to control his sabotage on S_1, S_2 and S_3 in (10') in a coordinated fashion so as to fake it through the entire verification test.

3rd form of Attack – Forging Z.

In this case, the hacker could return a forged value for the z term as if it were calculated from (5') to sabotage the calculation of (10'). However, it is practically impossible to do so because any forgery on the z values sabotaging S_1 will be routed through $\dot{S}_{fa}$ and $\dot{S}_{ha}$ before landing on (10'). The hacker would have no way to predict and control the impact of $\dot{S}_{fa}$ and $\dot{S}_{ha}$ during the 2nd iteration due to his null knowledge of g_a and g_b.

Moreover, neither could the hacker return a forged value for z as if it were from (8'). Imagine that the hacker faked some z values in (8') to give $\ddot{S}_1$ and $\ddot{S}_2$ that were seemingly good for the test of (10'). Since 2 alterations ($\ddot{S}_1$ and $\ddot{S}_2$) cannot satisfy a 3-way agreement (among S_1, S_2 and S_3) at the same time, the attack is essentially not possible.

D. Other Attack Consideration

Hacker trying to crack the private key e (2.1') would have to involve himself in the guesswork for the private data in the client's calculations of (7') and (9'). Take the first formulation of (2.1') for example, the hacker with the z values known to him from (5') and (8') would have to match the z values to the formulas in (7') and (9') and guess out the values for the r_{aij}, s_{aij} and t_{aij} terms for the calculation.

If we choose k =4, we will have 8 f_{aij}, 8 g_{aij} and etc. in (4'). That will give 32 z values in (5'). Suppose the r_{aij}, s_{aij} and t_{aij} terms all have values ranging from 1 to 15. Matching up the z values to the formulas in (7') and guessing the r_{aij}, s_{aij} and t_{aij} values for calculation of the formulas would cost

i) $C(32,8)\cdot 15^8$ searches for the calculation related to r_{aij}'s in (7')

ii) $C(24,8)\cdot 15^8$ searches related to t_{aij}'s in (7')

iii) 15^8 searches related to s_{aij}'s in (9').

Altogether the hacker will be running up against a search space of

$$C(32,8)\cdot 15^8 \cdot C(24,8)\cdot 15^8 \cdot 15^8 \cong 10^{41} \cong 2^{128}$$

Security strength by such search space is satisfactory.

E. Efficiency Consideration

The computational burden for the client this time is primarily due to (7') and (9'). There are 6 formulas of exponentiation to be evaluated. By the same analysis we did in the previous embodiment, the number of exponentiations to be carried out in (7') and (9') together is 48. As the exponents are 4-bit numbers, the worst case would reckon roughly 192 modular multiplications and the average case is roughly 96.

Compared with the 2048 multiplications in the regular 1024-bit RSA, this method gives the client device a saving factor of 10 or more on the CPU demand.

A number of references have been cited, the entire disclosures of which are incorporated herein by reference.